Learning union of k-testable languages

Statistical and symbolic language modeling project

Rania el Bouhssini, Martin Laville and Félix Jamet

January 12, 2019

Contents

1 Introduction

Unless explicitely specified, all definitions and algorithms in this document are coming from Linard *et al.* (2018), which will sometimes be refered to as "the paper".

We will present a possible implementation of those definitions and algorithms in a modular fashion, using Python3. Modular meaning here that we will implement concepts as they come. So if an `__init__` appears in the wild without its enclosing `class`, it's nothing to worry about.

The source of this document, an up-to-date version of it and the generated code are disponible on github[1].

[1] See `https://github.com/mooss/ktestable`.

2 k-testable languages

A k-testable language is a language that can be recognised by sliding a window of size k over an input. By definition, we have $k > 0$ since sliding a window of null or negative size would hardly make any sense.

A language is said to be k-testable in the strict sense (k-TSS) if it can be represented using a construct called a k-test vector. The necessary informations and operations on this construct will be implemented in the `ktestable` class, present in the file `tangled_ktestable.py`.

2.1 k-test vector

A k-test vector is a 4-tuple $Z = \langle I, F, T, C \rangle$:

— $I \in \Sigma^{k-1}$ is a set of allowed prefixes,
— $F \in \Sigma^{k-1}$ is a set of allowed suffixes,
— $T \in \Sigma^{k}$ is a set of allowed segments, and
— $C \in \Sigma^{<k}$ is a set of allowed short strings satisfying $I \cap F = C \cap \Sigma^{k-1}$.

We will refer to I, F, T and C respectively as the allowed prefixes, suffixes, infixes and short strings. Moreover, we will refer to $I \cap F$, *i.e.* the prefixes that are also suffixes as presuffixes. An intuitive way to formulate the constraint on short strings is that the short strings of length $k-1$ have to be presuffixes and vice versa.

This definition can be translated into an init.

`Init k-test vector:`

```
def __init__(self, prefixes, suffixes, infixes, shorts, k=None):
    self.k = len(next(iter(infixes))) if k is None else k
    self.prefixes = prefixes
    self.suffixes = suffixes
    self.infixes = infixes
    self.shorts = shorts
    self.ensure_correct_definition()
```

We then write `ensure_correct_definition` to make sure that the created k-test vector respects the conditions of the definition.

`Ensure correct definition:`

```
def ensure_correct_definition(self):
    def same_length(collection, reference_length):
        return all(map(lambda x: len(x) == reference_length, collection))

    errors = []
    if not same_length(self.prefixes, self.k - 1):
        errors.append('incorrect prefix length')
    if not same_length(self.suffixes, self.k - 1):
        errors.append('incorrect suffix length')
    if not same_length(self.infixes, self.k):
        errors.append('incorrect infix length')
    if not all(map(lambda x: len(x) < self.k, self.shorts)):
        errors.append('incorrect short string length')

    presuffixes = self.prefixes & self.suffixes
    shorts_len_k = set(filter(lambda x: len(x) == self.k - 1, self.shorts))
```

```
    if presuffixes != shorts_len_k:
        errors.append('short strings conditions not satisfied')

    if len(errors) >0:
        raise ValueError(', '.join(errors).capitalize() + '.')
```

2.2 k-test vectors as a partially ordered set

Let $\mathcal{T}_k$ be the set of all k-test vectors. A partial order $\sqsubseteq$ can be defined on $\mathcal{T}_k$ as follow:

$$\langle I, F, T, C\rangle \sqsubseteq \langle I', F', T', C'\rangle \iff I \subseteq I' \wedge F \subseteq F' \wedge T \subseteq T' \wedge C \subseteq C'$$

With this partial order, a union, an intersection and a symmetric difference can be defined on the k-test vectors $Z = \langle I, F, T, C\rangle$ and $Z' = \langle I', F', T', C'\rangle$.

First, we need to be able to check whether two `testable` are compatible, *i.e.* whether they have the same `k`.

`k-test vector compatibility:`

```
def ensure_compatibility(self, other):
    if self.k != other.k:
        raise ValueError('Incompatible k-test vectors: length mismatch (%d != %d)' %
                         (self.k, other.k))
```

2.2.1 Union ($\sqcup$)

$$Z \sqcup Z' = \langle I \cup I', F \cup F', T \cup T', C \cup C' \cup (I \cap F') \cup (I' \cap F)\rangle$$

We can see that the constraint on short strings $I \cap F = C \cap \Sigma^{k-1}$ is still respected because the short strings are updated with all the cases which could contradict it.

The implementation is a quite litteral translation if this definition.

`k-test vector union:`

```
def union(self, other):
    self.ensure_compatibility(other)
    prefixes = self.prefixes | other.prefixes
    suffixes = self.suffixes | other.suffixes
    infixes = self.infixes | other.infixes
    shorts = self.shorts | other.shorts |\
        (self.prefixes & other.suffixes) |\
        (self.suffixes & other.prefixes)
    return ktestable(prefixes, suffixes, infixes, shorts, k=self.k)
```

2.2.2 Intersection ($\sqcap$)

$$Z \sqcap Z' = \langle I \cap I', F \cap F', T \cap T', C \cap C'\rangle$$

Once again, the implementation is straightforward.

`k-test vector intersection:`

```
def intersection(self, other):
    self.ensure_compatibility(other)
    prefixes = self.prefixes & other.prefixes
    suffixes = self.suffixes & other.suffixes
```

```
    infixes = self.infixes & other.infixes
    shorts = self.shorts & other.shorts
    return ktestable(prefixes, suffixes, infixes, shorts, k=self.k)
```

2.2.3 Symmetric difference (△)

$$Z \triangle Z' = \langle I \triangle I', F \triangle F', T \triangle T', C \triangle C' \triangle (I \cap F') \triangle (I' \cap F) \rangle$$

Once more, it's only a matter of translating the set operations into python code.

`k-test vector symmetric difference`:

```
def symmetric_difference(self, other):
    self.ensure_compatibility(other)
    prefixes = self.prefixes ^ other.prefixes
    suffixes = self.suffixes ^ other.suffixes
    infixes = self.infixes ^ other.infixes
    shorts = self.shorts ^ other.shorts ^\
        (self.prefixes & other.suffixes) ^\
        (self.suffixes & other.prefixes)
    return ktestable(prefixes, suffixes, infixes, shorts, k=self.k)
```

2.2.4 Operators

Since the semantic of the three operations defined above are similar to those of sets, we create operators for them, matching the operators of `set`, the python builtin. That is to say `|` for union, `&` for intersection and `^` for symmetric difference.

`k-test vector operators`:

```
def __or__(self, other):
    return self.union(other)

def __and__(self, other):
    return self.intersection(other)

def __xor__(self, other):
    return self.symmetric_difference(other)
```

2.3 Measures

The theorem 3 of Linard *et al.* (2018) states that

> Any language that is a union of k-TSS languages can be identified in the limit from positive examples.

We will call "a union of k-TSS languages" a k-TSS-union. This Theorem means that when trying to learn a k-TSS-union from examples, the language will be learned at some point, having only seen a finite number of examples, even though the language might have an infinite number of examples.

It provides us with a baseline algorithm to learn a k-TSS-union. We consider each example as a language of its own and take the union of those examples. One problem of this algorithm is that it requires a great number of k-test vectors and will thus tend to be computationnaly expensive.

The solution to this problem is to consider it as a clustering problem by putting together similar vectors. The clustering algorithm will be seen later. Before this, there is a need to define a metric on k-test vectors, metric which will use the notion of cardinality.

2.3.1 Cardinality

The cardinality of a k-test vector $Z = \langle I, F, T, C \rangle$ is defined as:

$$|Z| = |I| + |F| + |T| + |C \cap \Sigma^{k-1}|$$

Once again, we see the influence of the short strings constraint since only the short strings of length less then $k-1$ are taken into account. Curiously, there is nothing in place to compensate for the presuffixes being counted twice. An alternative measure that takes this deduplication into account can be defined as:

$$|Z| = |I| + |F| + |T| + |C \cap \Sigma^{k-1}| - |I \cap F|$$

But we will still use the original definition.

`k-test vector cardinality`:

```
def cardinality(self):
    return len(self.prefixes) + len(self.suffixes) + len(self.infixes) +\
        sum(map(lambda x: 1 if len(x) < self.k - 1 else 0, self.shorts))

def __len__(self):
    return self.cardinality()
```

We also defined the operator `len`, since the meaning is similar to the builtin `len` of python sets.

2.3.2 Distance

The distance between two k-test vectors is the cardinality of their symmetric difference:

$$d(Z, Z') = |Z \triangle Z'|$$

It corresponds intuitively to the number of constituents that must be added or removed in order to go from one k-test vector to the other.

`k-test vector distance`:

```
def distance(self, other):
    return len(self ^ other)
```

2.4 Creation from an example

The provided `__init__` method can only construct a k-testable from its components. It's fairly easy to construct the minimal prefixes, suffixes, infixes and short strings necessary to detect an example e, or as the authors of the paper call it, a canonical k-test vector.

The prefixes and suffixes are simply the sets composed of the prefix and suffix of the example. The infixes can be defined by extracting all substring of length k. The only thing to be mindful of is the short strings condition and the case where $|e| < k-1$ (when there are no prefixes, only a short string). The $|e| = k$ case sorts itself out because in this situation, the example is just one presuffix.

`4-tuple from example`:

```
def ktest_tuple(example, k):
    if len(example) < k - 1:
        prefixes = set()
        suffixes = set()
        shorts = {example}
    else:
        prefixes = {example[:k-1]}
```

```
        suffixes = {example[-k+1:]}
        shorts = prefixes & suffixes

    infixes = {example[i:i+k] for i in range(0, len(example) - k + 1)}
    return (prefixes, suffixes, infixes, shorts)
```

We use this function to create a factory method for the `ktestable` class.

`Construct ktestable from example:`

```
@classmethod
def from_example(cls, example, k):
    return cls(*ktest_tuple(example, k), k)
```

2.5 Tests

We make some tests to ensure that the implementation works at least superficially as intended:

```
tests = {
    'invalid example': ({'aa'}, {'aa'}, {'aaaa'}, {'ada'}),
    'aa+': ktest_tuple('aaa', 3),
    'bb+': ktest_tuple('bbb', 3)
}
instanciations = {}

for name, parameters in tests.items():
    try:
        ktest = ktestable(*parameters)
        print('The creation of "%s": %s went well' % (name, parameters))
        instanciations[name] = ktest
    except ValueError as e:
        print('The creation of "%s": %s failed:\n -' % (name, parameters), e)


union = instanciations['aa+'] | instanciations['bb+']
intersection = instanciations['aa+'] & instanciations['bb+']
symmetric_difference = instanciations['aa+'] ^ instanciations['bb+']

print(union.prefixes)
print(intersection.prefixes)
print(symmetric_difference.prefixes)

print(union.distance(union))
print(union.distance(intersection))
print(len(union), len(intersection))
```

```
The creation of "invalid example": ({'aa'}, {'aa'}, {'aaaa'}, {'ada'}) failed:
 - Incorrect prefix length, incorrect suffix length, short strings conditions not satisfied.
The creation of "aa+": ({'aa'}, {'aa'}, {'aaa'}, {'aa'}) went well
The creation of "bb+": ({'bb'}, {'bb'}, {'bbb'}, {'bb'}) went well
{'aa', 'bb'}
set()
{'aa', 'bb'}
```

```
0
6
6 0
```

3 Efficient algorithm

The efficient algorithm presented in the paper creates one language per example and applies a hierarchical clustering algorithm to merge the languages two by two, if they are compatible.

In this part, we will first see how to find out if two languages are compatible (*i.e.* if their union is consistent). We will then present an alternative to the nearest-neighbour algorithm to produce a union of k-testable languages.

3.1 Union consistency definition

Before learning the union of languages, we need to ensure the union consistency between two k-test vectors Z and Z', *i.e.* the fact that the union of their languages should be the languages of their union. Linard *et al.*'s proposition 4 provides a way to do this.

Proposition 4 relies on padded prefixes and suffixes. A padded prefix is a prefix with an out-of-alphabet character $\bullet$ added at the beginning of its string. A padded suffix adds this character at the end of its string.

The idea is to create an oriented graph from the two k-test vectors, where a path starting from a prefix, ending in a suffix and passing through infixes will represent a word generated by the union of those k-test vectors. We will call this graph the consistency graph, and there are three aspects to it:

The vertices are the padded prefixes, the padded suffixes and the infixes:

$$V = \{\bullet u | u \in I \cup I'\} \cup \{u \bullet | u \in F \cup F'\} \cup T \cup T'$$

The edges are drawn from one vertex to the other if the suffix of size $k-1$ of the first vertex is equal to the prefix of size $k-1$ of the second vertex:

$$E = \{(au, ub) \in V \times V | a, b \in \Sigma \cup \{\bullet\}, u \in \Sigma^{k-1}\}$$

The colors are reflecting whether a vertex is "endemic" to one vector:

— a red vertex is endemic to Z,
— a blue vertex is endemic to Z', and
— a white vertex is endemic to both.

A vertex v is endemic to a vector $X = \langle I, F, T, C\rangle$ compared to another vector $X' = \langle I', F', T', C'\rangle$ if it appears only in X. More formally, it is endemic if the following holds:

$$\begin{cases} u \in I \setminus I' & \text{if } v = \bullet u \\ u \in F \setminus F' & \text{if } v = u\bullet \\ v \in T \setminus T' & \text{otherwise} \end{cases}$$

The paper shows that the union consistency is ensured if and only if there exists no path between a red vertex and a blue vertex. A path between red and blue vertices means that a word out of both languages emerges in the union, which is precisely what we want to avoid.

We will first compute the consistency graph and then test the union consistency. Both of these operations will be implemented into their own method of `ktestable`.

3.2 Consistency graph

For brevity and sanity's sake, we will use a library to do operations on graphs. We have chosen NetworkX[1] since it has proved to be easier to install than the alternatives.

We suppose NetworkX has already been imported like this:

```
import networkx as nx
```

The method `consistency_graph` constructs the graph and consists of three parts ;

`Construct consistency graph:`

```
def consistency_graph(self, other):
    <<Vertices construction>>

    <<Edges construction>>

    <<Graph assembling>>
```

We construct the vertices but rather than using padded prefixes and suffixes, we prepend the letters `P` and `S` to the prefixes and the suffixes, respectively. Those letters allow us to distinguish between presuffixes. It is indeed possible to have a presuffix in the union but if the prefix is blue, then the suffix might be red and if we do not distinguish presuffixes, we will not be able to have the right result when searching for multicolor paths in the graph.

`Vertices construction:`

```
prefixes = {'P' + el for el in self.prefixes | other.prefixes}
suffixes = {'S' + el for el in self.suffixes | other.suffixes}
infixes = {el for el in self.infixes | other.infixes}
```

There are only three ways in which an edge can form between two vertices:

— a prefix can connect to an infix,
— an infix can connect to another infix, and
— an infix can connect to a suffix.

`Edges construction:`

```
edges = {(pre, inf) for pre in prefixes for inf in infixes
         if pre[1:] == inf[:-1]}
edges.update({(left, right) for left in infixes for right in infixes
              if left[1:] == right[:-1]})
edges.update({(inf, suf) for inf in infixes for suf in suffixes
              if inf[1:] == suf[1:]})
```

Since we are only interested by the paths between vertices, we construct the graph from the edges only, thus leaving out isolated vertices. In any case, there should not be isolated vertices because the k-test vectors are supposed to be well-constructed.

`Graph assembling:`

```
graph = nx.DiGraph()
graph.add_edges_from(edges)
return graph
```

[1] See `https://networkx.github.io/documentation/stable/install.html`.

3.3 Union consistency implementation

This part is mostly a matter of using the consistency graph to search for multicolor paths.

`Union consistency`:

```
def is_union_consistent_with(self, other):
    <<Path research>>

    <<Paths analysis>>
```

We compute only the red and blue vertices, we do not need the white. As has been done before, we prepend a `P` to prefixes and an `S` to suffixes. We then search for a path, using the fact that searching for a path between reds and blues is akin to find a transitive closure and examine the reachability of red and blue vertices with respect to one another.

`Path research`:

```
reds = {'P' + el for el in self.prefixes - other.prefixes} |\
    {'S' + el for el in self.suffixes - other.suffixes} |\
    self.infixes - other.infixes
blues = {'P' + el for el in other.prefixes - self.prefixes} |\
    {'S' + el for el in other.suffixes - self.suffixes} |\
    other.infixes - self.infixes

graph = self.consistency_graph(other)
closure = nx.algorithms.dag.transitive_closure(graph)
red_reachable = {neighbour for red in reds if red in closure for neighbour in
 ↪  closure.adj[red]}
blue_reachable = {neighbour for blue in blues if blue in closure for neighbour in
 ↪  closure.adj[blue]}
```

Finally, we only have to check if red vertices are reachable from blue vertices and vice versa.

`Paths analysis`:

```
return red_reachable.isdisjoint(blues) and blue_reachable.isdisjoint(reds)
```

3.4 Distance chain algorithm

Because we have had difficulties understanding the nearest-neighbour chain algorithm, we decided to roll our own algorithm to cluster k-test vectors. We dubbed it the distance chain algorithm.

The core idea is to provide an easy access to nearest-neighbours by storing edges in a pre-sorted data structure, the "distance chain".

First, some vocabulary:

A vertex is here a k-test vector.

An edge is a couple of distinct vertices, with their distance.

A distance link is a way to store some edges common to to one particular vertex, the left vertex, the other vertices are called the right vertices and are stored in a list, the neighbour list.

A distance chain is a list of distance links without edge duplication.

All distance links are sorted according to distance and the distance chain is sorted according to the distance of the closest vertices, *i.e.* the first vertex of the neighbour list.

There are two main intuitions:

— When the two closest vertices are not mergeable (because they are not union compatible), it is easy to discard them and find the next two closest vertices.
— After the two closest vertices are found, a lot of now useless edges can be discarded by throwing away the distance link whose left vertex is one of them.

A drawback is that there will remain edges to throw away, those whose right vertex was one of the two merged vertices. Our solution is to keep track of the vertices that were already merged and to discard them when they come up.

Our implementation uses `namedtuple`, exported like this:

```
from collections import namedtuple
```

We have separated our implementation into five parts which will be explained one after the other. The first part is only about the initialisation of data structures, among which sits the main one, the distance chain. The other parts are contained in a `while` loop and their goal is to deplete this distance chain.

`Distance chain algorithm`:

```
def learn_ktest_union(examples, k):
    <<Initialiase data structures>>

    while True:
        <<Find the closest mergeable vertices>>

        <<Update merge record>>

        <<Cleanup distance chain>>

        <<Update distance chain>>
```

3.4.1 Initialising the data structures

There are three variables whose role is to record the progression of the algorithm:

— `ktest_vectors` contains the initial vectors created from the examples and will be updated with the new merged vectors.
— `indexes` keeps tracks of all the merges that happened.
— `already_merged` records whether a vector was already merged with another.

We also construct the initial distance chain and we make sure it is sorted.

`Initialiase data structures`:

```
ktest_vectors = [ktestable.from_example(ex, k) for ex in examples]
indexes = list(range(len(ktest_vectors)))
already_merged = [False] * len(ktest_vectors)

distance_link = namedtuple('d', 'neighbours left')
distance_chain = []

for left in range(0, len(ktest_vectors) - 1):
    neighbours = []
    for right in range(left + 1, len(ktest_vectors)):
        neigh = (ktest_vectors[left].distance(ktest_vectors[right]), right)
        neighbours.append(neigh)
    neighbours.sort()
```

```
    distance_chain.append(distance_link(neighbours=neighbours, left=left))
distance_chain.sort()
```

3.4.2 Finding the closest mergeable vertices

The idea here is to loop until the closest mergeable vertices are found. When the closest vertices are not mergeable (because the right vertex was already merged or because they are not union consistent), we throw the edge away and try the next one.

We make sure that the distance chain stays sorted and we remove empty links. The exit point of the function is also here, the result is returned when the distance chain has been exhausted.

`Find the closest mergeable vertices`:

```
while True:
    if len(distance_chain) == 0:
        return [(ktest_vectors[x], indexes[x])
                for x, merged in enumerate(already_merged) if not merged]

    left, (dist, right) = distance_chain[0].left, distance_chain[0].neighbours[0]
    if not already_merged[right] and\
       ktest_vectors[left].is_union_consistent_with(ktest_vectors[right]):
        break

    del distance_chain[0].neighbours[0]
    if len(distance_chain[0].neighbours) == 0:
        del distance_chain[0]

    distance_chain.sort()
```

3.4.3 Updating the merge record

As a result of the previous block, `left` and `right` are now the closest mergeable vertices. We record the structure of the merge into `indexes`, the union into `ktest_vectors`, and the merge into `already_merged`.

`Update merge record`:

```
indexes.append((indexes[left], indexes[right], dist))
ktest_vectors.append(ktest_vectors[left] | ktest_vectors[right])
already_merged.append(False)
already_merged[left] = already_merged[right] = True
```

3.4.4 Cleaning up the distance chain

The cleanup highlights the advantage of the algorithm, the fact that a chunk of the distance chain can be quickly removed by discarding the links whose left vertex is `left` or `right`.

`Cleanup distance chain`:

```
del distance_chain[0]
for i in range(len(distance_chain)):
    if distance_chain[i].left == right:
        del distance_chain[i]
        break
```

3.4.5 Updating the distance chain

Finally, we construct the distance link corresponding to the k-test union that was just created all the while making sure that the distance chain is still sorted. It would be more efficient to simply insert the new distance link at its place in the distant chain rather than sorting the list all over again, but we wanted to put the emphasis on readability.

Update distance chain:

```
neighbours = []
for right, merged in enumerate(already_merged[:-1]):
    if not merged:
        neighbours.append((ktest_vectors[-1].distance(ktest_vectors[right]), right))

neighbours.sort()
distance_chain.append(distance_link(neighbours=neighbours, left=len(ktest_vectors) - 1))
distance_chain.sort()
```

3.5 Tests

Some basic tests based on examples from the paper.

Union consistency test:

```
examples = {
    'z3': ({'ab'}, {'bc'}, {'abc', 'bca', 'cab'}, {}),
    'z4': ({'cb'}, {'ba'}, {'cba', 'bac', 'acb'}, {}),
    'z5': ({'ab'}, {'ba'}, {'abb', 'bbb', 'bba'}, {}),
    'z7': ({'ab'}, {'ba'}, {'abb', 'bbb', 'bba'}, {}),
}

instances = {iden: ktestable(*params) for iden, params in examples.items()}

print(instances['z5'].is_union_consistent_with(instances['z7']))
print(instances['z3'].is_union_consistent_with(instances['z4']))
print(instances['z3'].is_union_consistent_with(instances['z7']))

print()
print(ktestable.from_example('baba',
 ↪  3).is_union_consistent_with(ktestable.from_example('bababbc', 3)))

with open('dataset/paper.txt') as datasetfile:
    dataset = [line.rstrip('\n') for line in datasetfile]

res = learn_ktest_union(dataset, 3)
clusters = list(map(lambda x: x[1], res))
print(clusters)
print(len(clusters))
```

```
True
True
False

True
[((4, 6), 1), (0, 7), ((3, 5), 2)]
3
```

4 About proposition 4

As discussed in section 3.1, proposition 4 provides a way to find out whether the union of the languages is the language of the union, by building a consistency graph. We propose here a way to simplify this computation.

There are only two ways to go from a red to a blue component ; via a direct path and via an indirect path. We only consider paths from blue to red but the same applies to paths from red to blue. A direct path is when an edge exists between a blue and a red vertex, whereas an indirect path goes from to a blue to a red vertex, while going through n white vertices, $n \in \mathbb{N}_{>0}$.

A direct path is easy to detect and occurs when a superposition is possible between a red component and a blue component. We define the conditions for superposition to occur in the table 4.1, with $c \in \Sigma$ and $u, v \in \Sigma^{k-1}$.

Left	Right	Condition for superposition
Prefix I	Infix $T = uc$	$I = u$
Infix $T = cv$	Infix $T' = uc$	$v = u$
Infix $T = cv$	Suffix F	$v = F$

Table 4.1: Conditions for superpositions between prefixes, infixes and suffixes

An indirect path is more tricky to detect, because it can pass through any number of white components. Our approach is based on the observation that to pass from blue to red, the only white components an indirect path can go through are infixes. We consider that white infixes who have a superposition with blue prefixes or infixes are "de facto" blue infixes and that a superposition between a de facto blue infix and a white infix is another de facto blue infix. With that in mind, an indirect path from blue to red exists when a superposition is possible between a de facto blue infix and a red infix or suffix.

We provide no demonstration for our claim but we hope that one can be derived from the preexisting demonstrations in the paper. However, we do provide an implementation for this optimised way of computing the union consistency, using this time a hand rolled transitive closure rather than a graph library.

4.1 De facto colored states identification

We can reformulate the definition of "de facto blue states" that we gave earlier as finding all the white infixes reachable from blue states. This is simply a matter of computing a transitive closure.

For this, we will make use of the defaultdict class:

```
from collections import defaultdict
```

Our implementation of transitive closure is a function that can be separated in three parts.

`String transitive closure:`

```
def string_transitive_closure(starting_components, infixes):
    <<Prefixes of infixes>>

    <<Initial closure>>

    <<Transitive closure>>
```

`prefdict` is a mapping from the prefixes of the infixes to the infixes. It allows us to quickly find superpositions.

`Prefixes of infixes:`

```
prefdict = defaultdict(set)
for inf in infixes:
    prefdict[inf[:-1]].add(inf)
```

The initial states in the closure are those who have a superposition with the starting components.

`Initial closure`:

```
closure = set()
for pref in starting_components:
    if pref in prefdict:
        closure.update(prefdict.pop(pref))
```

To complete the transitive closure, we iterate on all the elements in the closure and add to it all the new superpositions.

`Transitive closure`:

```
result = set()
while closure:
    el = closure.pop()
    result.add(el)
    if el[1:] in prefdict:
       closure.update(prefdict.pop(el[1:]))
return result
```

4.2 Union consistency v2

We can now implement the new union consistency test by getting the colored components and checking for direct and indirect paths between red and blue.

`Union consistency v2`:

```
def is_union_consistent_with(self, other):
    <<Colored components>>

    <<Direct paths>>

    <<Indirect paths>>
```

For each color, we compute the infixes, as well as the potential starting and stopping points of paths. The potential starting points are the prefixes and the suffixes of infixes and the potential stopping points are the suffixes and the prefixes of infixes.

`Colored components`:

```
red_infixes = self.infixes - other.infixes
red_start = self.prefixes - other.prefixes
red_start.update(inf[1:] for inf in red_infixes)
red_stop = self.suffixes - other.suffixes
red_stop.update(inf[:-1] for inf in red_infixes)

blue_infixes = other.infixes - self.infixes
blue_start = other.prefixes - self.prefixes
blue_start.update(inf[1:] for inf in blue_infixes)
```

```
blue_stop = other.suffixes - self.suffixes
blue_stop.update(inf[:-1] for inf in blue_infixes)
```

If a direct path is found, we can stop early without having to search for indirect paths, which is more costly.

`Direct paths:`

```
if blue_start & red_stop or red_start & blue_stop:
    return False
```

If there were no direct paths, the transitive closure function is used to find the de facto blue states and the de facto red states. Finally, if no superposition exists between de facto red states and blue states and vice-versa, this means that there is no path between those two colors, and therefore that the union is consistent.

`Indirect paths:`

```
white_infixes = self.infixes & other.infixes
de_facto_red = string_transitive_closure(red_start, white_infixes)
de_facto_blue = string_transitive_closure(blue_start, white_infixes)

de_facto_red = {el[1:] for el in de_facto_red}
de_facto_blue = {el[1:] for el in de_facto_blue}

return not(de_facto_blue & red_stop) and not(de_facto_red & blue_stop)
```

This new union consistency is present in the file `tangled_ktestable_v2`.

4.3 Measurements

While this way to look at the union consistency problem still relies on graph, it has intuitively the advantage of being less expensive to compute since it can identify an early exit condition and when that condition is not met, it compute the transitive closure only on a small subset of the graph.

We propose to measure experimentally the speedup provided.

4.3.1 Union consistency

We measure the speed of execution of the two version of the union consistency test on all the possible combinations of the 300 first examples of the printjob dataset.

```
from itertools import combinations
from tangled_ktestable import ktestable as ktestable_v1
from tangled_ktestable_v2 import ktestable as ktestable_v2

with open('dataset/printjob.txt') as datasetfile:
    dataset = [line.rstrip('\n') for line in datasetfile]

dataset = dataset[:300]
instances_v1 = [ktestable_v1.from_example(ex, 3) for ex in dataset]
instances_v2 = [ktestable_v2.from_example(ex, 3) for ex in dataset]

print('v1')
%timeit -n 5 -r 5 [left.is_union_consistent_with(right) for left, right in
 ↪  combinations(instances_v1, 2)]
print('v2')
%timeit -n 5 -r 5 [left.is_union_consistent_with(right) for left, right in
 ↪  combinations(instances_v2, 2)]
```

```
v1
14.4 s ± 185 ms per loop (mean ± std. dev. of 5 runs, 5 loops each)
v2
280 ms ± 170 µs per loop (mean ± std. dev. of 5 runs, 5 loops each)
```

We can see that this optimisation is about two orders of magnitude faster than the original.

4.3.2 Union learning

Below is an extract of a line by line profile, obtained with the module `line_profiler`[1], on the initial version of `ktestable`, while learning the printjob dataset with $k = 3$.

```
#Line# % Time  Line Contents
#================================
145      0.0   for left in range(0, len(ktest_vectors) - 1):
146      0.0      neighbours = []
147      1.4      for right in range(left + 1, len(ktest_vectors)):
148     40.2         neigh = (ktest_vectors[left].distance(ktest_vectors[right]), right)
149      1.6         neighbours.append(neigh)
150      0.4      neighbours.sort()
151      0.0      distance_chain.append(distance_link(neighbours=neighbours, left=left))
152      0.0   distance_chain.sort()
153
154      0.0   while True:
155      0.0      while True:
156      0.0         left, (dist, right) = distance_chain[0].left,
 ↪  distance_chain[0].neighbours[0]
157      0.0         if not already_merged[right] and\
158      3.2            ktest_vectors[left].is_union_consistent_with(ktest_vectors[right]):
159      0.0            break
...      ...      ...
181      0.0      neighbours = []
182      4.5      for right, merged in enumerate(already_merged[:-1]):
183      4.2         if not merged:
184     40.4            neigh = (ktest_vectors[-1].distance(ktest_vectors[right]), right)
185      1.7            neighbours.append(neigh)
```

In the grand scheme of things, we can see that the union consistency test, even when using the initial version, is not the bottleneck. The bottleneck appears to be the distance computation.

Indeed, when we look at the time it takes to learn from the printjob dataset, the difference can barely be seen between the two versions:

```
from tangled_ktestable import learn_ktest_union as learn_ktest_union_v1
from tangled_ktestable_v2 import learn_ktest_union as learn_ktest_union_v2

with open('dataset/printjob.txt') as datasetfile:
    dataset = [line.rstrip('\n') for line in datasetfile]

print('v1')
%time learn_ktest_union_v1(dataset, 3)
print('v2')
%time learn_ktest_union_v2(dataset, 3)
```

[1] See `https://pypi.org/project/line_profiler/`

```
v1
CPU times: user 35.7 s, sys: 155 ms, total: 35.9 s
Wall time: 36 s
v2
CPU times: user 35 s, sys: 237 ms, total: 35.3 s
Wall time: 35.7 s
```

But if we use a larger value of k, the optimised version makes a big difference:

```
print('v1')
%time learn_ktest_union_v1(dataset, 6)
print('v2')
%time learn_ktest_union_v2(dataset, 6)
```

```
v1
CPU times: user 13min 5s, sys: 569 ms, total: 13min 5s
Wall time: 13min 7s
v2
CPU times: user 1min 46s, sys: 196 ms, total: 1min 46s
Wall time: 1min 47s
```

This is intuitively understandable, because since the alphabet of the printjob dataset only has 6 characters, a lot of segments of size k will be repeated for very small values of k. This repetition does not matter because the components are stored in sets. Therefore, the graph will stay small, and it will be fast to go through it. For other scenarios, with larger values of k and larger alphabets, our proposition can make a big difference.

5 Coherence tests

We create the test script `coherence_tests.py` in order to make sure that we have the same results across all our methods. This script takes two arguments, to specify the dataset to use and the consistency method.

`Argument handling`:

```
from operator import itemgetter
import argparse
parser = argparse.ArgumentParser(description='Learn a k-test union from a dataset')
parser.add_argument('dataset', type=str)
parser.add_argument('method', type=str, choices=['graph', 'de_facto'])
parser.add_argument('k', type=int, default=3, nargs='?')
args = parser.parse_args()
```

We load the learning function from the right file:

```
if args.method == 'graph':
    from tangled_ktestable import learn_ktest_union
else:
    from tangled_ktestable_v2 import learn_ktest_union
```

We load the dataset:

```
with open(args.dataset) as datasetfile:
    dataset = [line.rstrip('\n') for line in datasetfile]
```

We learn the clusters and print them:

```
union = learn_ktest_union(dataset, args.k)
clusters = list(map(itemgetter(1), union))
print(clusters)
print(len(clusters))
```

We can now use this script to check if our implementations return the same results on the same dataset. We use the printjob dataset since it is the most comprehensive dataset we have.

```
python3 coherence_tests.py dataset/printjob.txt graph > graph
python3 coherence_tests.py dataset/printjob.txt de_facto > de_facto
wc -l graph de_facto
echo difference:
diff graph de_facto
```

```
   2 graph
   2 de_facto
   4 total
difference:
```

6 Conclusion and discussion

We have detailed an alternative to the nearest-neighbour chain algorithm and proposed an amelioration of proposition 4. We initially set out to find an approximation to the rate of generalisation of the union of k-test vectors, with the aim of completing the hierarchical clustering when no union consistent vectors remained. We did not have enough time to implement it, but we can quickly explain the gist of it here.

We think that a good proxy for the rate of generalisation is the number of ways to form a new word, that is to say the number of paths between red and blue in the consistency graph. Using what we have done in part 4, we can make a baseline for this. The number of direct paths is trivial to count. Once more, the indirect paths prove themselves more difficult to handle because there can be loops between de facto colored vertices. We propose to simply multiply the number of entry points with the number of exit points. We add to this the number of direct paths and we have our measure. This is a bit handwavy but it might make a good enough baseline nevertheless.

Once we are at the stage where all vectors are union inconsistent, we continue the clustering with this measure. The usefulness of this approach might be tested with the following experiment:

— taking the smallest union obtained without using this measure with a small value of k,
— learning the union on the same dataset, this time with a larger value of k,
— using the measure to extend the clustering until the same union size is reached, and
— comparing the two languages obtained.

In the end, we think that k-testable languages are a simple yet deep concept with a lot of untapped potential.

7 Sources

1. Linard, A., de la Higuera C., Vaandrager F.:Learning Unions of k-Testable Languages, (2018): https://arxiv.org/abs/1812.08269